Comparison of the performance of parallelizing the K-Means algorithm using MPI, OMP, and a hybrid version

High Performance Computing (HPC) Project

Edoardo Maines (232226)

June 21, 2024

University of Trento

1 Introduction

In a world where data production is increasing day by day the need for data analytic method to extract information from the vast disposition of data it's growing faster. Among the technique used for analyse data, clustering is one of the most used. The objective of clustering is to subdivide a dataset into subcategories based on the data characteristics, however analizing a large quantity of data can be challenging in terms of execution time and resources. To cope for this problem there are techniques to parallelize the execution of these algorithms and improve performances.

In this paper, the implementation of a clustering algorithm written in C language is presented, using MPI (Message Passing Interface) and OMP (Open Multi-Processing) to parallelize the code. The scope of this work is to compare the performances between the serial and the parallel executions. Particularly, the execution time and the precision of the different implementations of the algorithms (OMP, MPI) are compared to better understand which one is more efficient. In the end, a third implementation that consists of a hybrid approach (MPI+OMP) was finalized to observe if the union of both methodologies could improve the performances.

The work it's been implemented and tested using the server cluster belonging to the university of Trento.

2 State of the Art

Clustering is a data analysis technique widely used in a lot of disciplines as biology, computer science, statistics and much more. There are different kind of clustering algorithms among which hierarchical clustering, k-means clustering and Gaussian Mixture Model clustering.

Hierarchic clustering subdivide the data in hierarchic groups applying subsequential division till each group has only one element building this way a dendogram to represent the data distribuition and possible clusters as shown in [3].

K-means clustering is a partitioning technique that attempts to divide the data into a fixed number of groups, so that the variance within the groups is minimal, while Mixed model clustering is a technique that uses probability models to associate data with groups.

Some clustering algorithms were designed for clustering large amounts of data. For example, clustering based on grid patterns divides the data space into a grid, where each cell represents a group. This technique is particularly useful for analyzing geospatial data.

Different state-of-the-art algorithms for parallelizing the k-means with MPI [4] [**Yang**] and OMP [2] [1] were analyzed.

3 Parallelization methods

3.1 MPI (Message Passing Interface)

MPI is a parallel programming library widely used to develop distributed and parallel applications on computer clusters. It enables communication between processes, allowing them to exchange data and coordinate operations synchronously or asynchronously. The main features of MPI include message management, synchronization, data distribution, and scalability across distributed systems. MPI provides a set of functions for communication and synchronization between processes.

- **`MPI_Init()` e `MPI_Finalize()`**

  The functions `MPI_Init()` and `MPI_Finalize()` are used to initialize and terminate the MPI environment. `MPI_Init()` is called at the beginning of the MPI program to initialize the MPI environment and establish communication between processes. `MPI_Finalize()` is called at the end of the program to clean up and release resources used by the MPI environment.

- **`MPI_Allreduce()`**

  The function `MPI_Allreduce()` performs a global reduce operation on all processes in the MPI communicator and returns the result to all processes. It is useful when you need to combine the results of each process into a single variable.

- **`MPI_Reduce()`**

  The function `MPI_Reduce()` performs a reduce operation on all processes in the MPI communicator and returns the result to the root process. This is similar to `MPI_Allreduce()`, but the result is returned only to the root process.

- **`MPI_Barrier()`**

  The function `MPI_Barrier()` it is used to synchronize all processes in the MPI communicator. Each process waits until all other processes in the communicator have reached the synchronization point before continuing.

3.2 OMP (Open Multi-Processing)

OpenMP is a parallel programming API for programming on systems with shared memory, such as multicore systems. It allows you to perform operations in parallel using pragma directives within the C/C++ source code. OpenMP simplifies the parallelization of loops, code regions, and parallel tasks, allowing you to maximize the processing potential of modern multicore processors.

3.2.1 Direttive pragma

OpenMP pragma directives are special commands that are interpreted by the compiler to guide the parallelization process. The pragma directives starts with `#pragma omp` followed by a specific instruction.

- **`#pragma omp parallel`**
  The directive `#pragma omp parallel` is used to create a team of threads, where the immediately following block of code will be executed by all threads in the team.

- **`#pragma omp parallel for`**
  The directive `#pragma omp parallel for` is used to parallelize a loop, allowing threads to iterate through the loop in parallel.

  - **`#pragma omp parallel for collapse (2)`**
    The directive `#pragma omp parallel for collapse (2)` is used to parallelize two nested loops, allowing threads to iterate in parallel. This directive can improve performance when you have nested loops.

- **`#pragma omp atomic`**
  The directive `#pragma omp atomic` is used to perform an operation atomically, avoiding concurrency between threads.

- **`#pragma omp single`**
  The directive `#pragma omp single` is used to execute a block of code from a single thread, ensuring that it runs only once within a team of threads.

4 Algorithm

The chosen clustering algorithm is k-means clustering, a partitioning algorithm that divides the data into a fixed number of groups, minimizing the variance within the groups. K-means clustering was selected due to its simplicity and widespread adoption in the clustering community. Moreover, k-means is highly parallelizable, which makes it particularly suitable for leveraging MPI and OMP libraries.

4.1 K-Means

The k-means algorithm is a widely used clustering method for grouping unlabeled data into a fixed number of clusters. Its goal is to divide a data set into homogeneous groups so that the points within each group are similar to each other and different from the points in the other groups. Refer to Algorithm 1 at the end of the paper for the pseudocode.

The k-means algorithm proceeds as follows:

1. **Initialization of Centroids:** Initialize cluster centroids randomly or using a deterministic method.

2. **Assignment of Data Points:** For each data point, calculate its distance to each centroid. Assign each data point to the nearest centroid.

3. **Update of Centroids:** Compute new centroids based on the mean of data points assigned to each centroid. Repeat until convergence criteria are met (e.g., centroids stabilize or maximum iterations reached).

4. **Convergence Check:** Monitor changes in centroids between iterations. Stop when centroids no longer change significantly or after a fixed number of iterations.

4.2 Parallelization of the k-means algorithm with MPI

The k-means algorithm is parallelized using the Message Passing Interface (MPI), a framework for distributed computing across multiple nodes in a cluster. Refer to Algorithm 2 at the end of the paper for the pseudocode.

The parallelization using MPI proceeds as follows:

1. **Initialization of Centroids:** Cluster centroids are initialized randomly. Process 0 is responsible for this. Centroids are distributed to other processes using MPI communication functions.

2. **Data Partitioning:** Dataset is partitioned among MPI processes. Each process handles a subset of the data.

3. **Iteration of k-means:** Each process computes distances between its data points and centroids. Processes assign points to the nearest centroids. Local statistics are computed and communicated back to process 0.

4. **Centroid Update:** Process 0 computes global averages and updates centroids. Updated centroids are broadcasted to all processes.

5. **Convergence Check:** Algorithm iterates until convergence criteria are met.

4.3 Parallelization of the k-means algorithm with OMP

The k-means algorithm is parallelized using OpenMP (OMP), a framework for shared-memory parallelism on multi-core processors. Refer to Algorithm 3 at the end of the paper for the pseudocode.

The parallelization using OMP proceeds as follows:

1. **Shared-memory Execution:** Data is shared among threads, and each thread operates on a subset.

2. **Directive-based Parallelism:** Compiler directives specify parallel sections of code. Distances computation, point assignment, and centroid updates are parallelized.

3. **Centroid Update:** Threads collaborate to update centroids. Synchronization mechanisms ensure consistency.

4. **Iteration and Convergence:** Algorithm iterates until convergence criteria are met.

4.4 Hybrid MPI-OMP Implementation of the k-means algorithm

The hybrid approach combines the strengths of MPI and OpenMP to leverage both distributed and shared-memory parallelism for improved performance in large-scale clustering tasks.

The parallelization using the combination of MPI and OMP proceeds as follows:

1. **Initialization of Centroids:** Process 0 initializes cluster centroids randomly and broadcasts them to all MPI processes.
2. **Data Partitioning:** MPI processes divide the dataset into partitions, with each process handling a subset of data. Within each MPI process, OpenMP directives are used to parallelize computations across multiple threads.
3. **Parallel Computation:** Each MPI process uses OpenMP to parallelize the computation of distances and assignment of data points to centroids. Local updates to centroids are performed in parallel within each MPI process using OpenMP directives.
4. **Communication and Synchronization:** MPI communication is used to exchange centroid updates and synchronize global centroids across all MPI processes.
5. **Convergence Check:** Iterative steps continue until convergence criteria are met, similar to the MPI and OpenMP implementations.

4.5 Comparison of Approaches

Each parallelization approach for the k-means algorithm—MPI, OpenMP (OMP), and the hybrid MPI-OMP approach—brings unique strengths and considerations to the table.

MPI excels in distributed memory environments, providing high scalability across clusters of computers. It allows for independent operation of each MPI process, making it suitable for diverse hardware architectures and heterogeneous systems. However, MPI programming can be complex due to explicit message passing and synchronization requirements, and it may suffer from communication overhead, especially on smaller clusters or for less parallelizable tasks.

OpenMP, on the other hand, leverages shared memory parallelism within a single node, which simplifies programming compared to MPI. It efficiently utilizes multi-core processors, minimizing communication overhead and synchronization costs by accessing shared data structures directly. However, OpenMP is limited to multi-core nodes and does not scale well across distributed systems. Careful management of shared memory is crucial to avoid race conditions and ensure data consistency.

The hybrid MPI-OMP approach combines the strengths of both models. It harnesses MPI for inter-node communication, load balancing, and scalability across distributed systems. Within each MPI process, OpenMP directives exploit shared-memory parallelism to maximize intra-node performance. This hybrid approach offers flexibility and optimizes resource utilization, balancing computational load across nodes and cores for efficient large-scale clustering tasks. However, it requires expertise in both MPI and OpenMP programming paradigms and careful design to manage communication overhead and synchronization effectively.

In summary, while MPI excels in scalability and control across distributed environments and OpenMP efficiently utilizes multi-core processors within nodes, the hybrid MPI-OMP approach bridges these strengths. It effectively combines distributed and shared-memory parallelism to achieve optimized performance for large-scale k-means clustering tasks, but it demands careful implementation and tuning to mitigate potential complexities and overhead.

4.6 Performance evaluation

As the algorithm runs, various performance metrics including I/O performance, k-means execution time, and total algorithm time are measured. These metrics serve to evaluate the effectiveness of parallelization in accelerating the algorithm's execution compared to its serial counterpart.

Parallelization techniques allow efficient utilization of both distributed and shared memory computing resources, enhancing scalability and performance of the k-means algorithm, particularly on large datasets. This study explores parallelization strategies across different computing architectures, leveraging multiple nodes or cores to achieve optimal computational efficiency. Specifically, the evaluation includes testing with three distinct datasets, varying the number of clusters (K values of 3, 6, and 9), and examining the impact of the number of processes/threads, ranging from 2 to 10.

To facilitate clear comparison and visualization of these performance metrics, two key metrics are computed: Speedup, representing the ratio of serial execution time to parallel execution time, and Efficiency, which measures the utilization of parallel resources relative to the ideal scenario.

The speedup is calculated as follows:

$$S = \frac{T_{serial}}{T_{parallel}}$$

where T_{serial} represents the time taken by the algorithm in serial, while $T_{parallel}$ represents the time taken by the parallelized implementation. While the efficiency of the program in parallel:

$$E = \frac{S}{p} = \frac{T_{serial}}{T_{parallel} \cdot p}$$

where p is the number of cores available. More specifically, in the cases of parallelization using MPI and OMP, p represents the number of processes and the number of threads, respectively.

In the case of the hybrid approach, the term p, indicating the number of processes used by MPI, is multiplied by a term t, representing the number of threads used by OMP (p * t), as shown in the following equation.

$$E = \frac{S}{(p \cdot t)} = \frac{T_{serial}}{T_{parallel} \cdot (p \cdot t)}$$

The resulting values for these metrics are shown in Tables 1, 2, and 3, respectively for the MPI, OMP, and hybrid approaches. The tables also include a comparison between the different datasets for K equal to 3, 6 and 9.

Comparison of MPI Performance Metrics for Different K Values						
Number of Processes	**K=3**		**K=6**		**K=9**	
	Efficiency	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**
2	0.860483	1.720966	0.862711	1.725423	0.729479	1.458958
3	0.716549	2.149648	0.727735	2.183205	0.603513	1.810539
4	0.581231	2.324923	0.623718	2.494871	0.552503	2.210012
5	0.538460	2.692300	0.554441	2.772206	0.580493	2.902466
6	0.458777	2.752661	0.495604	2.973626	0.501304	3.007824
7	0.400060	2.800418	0.452750	3.169251	0.459731	3.218116
8	0.354858	2.838863	0.414930	3.319439	0.430395	3.443158
9	0.349623	3.146608	0.382743	3.444690	0.399318	3.593862
10	0.315235	3.152354	0.350588	3.505878	0.373334	3.733344

Table 1: Comparison of MPI Performance Metrics for Different K Values

Comparison of OMP Performance Metrics for Different K Values						
Number of Processes	**K=3**		**K=6**		**K=9**	
	Efficiency	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**
2	0.989542	1.979084	0.998924	1.997847	0.970196	1.940392
3	0.792414	2.347241	0.825223	2.475669	0.815737	2.447212
4	0.637823	2.551294	0.751991	3.007964	0.698491	2.793966
5	0.559797	2.798987	0.627570	3.137852	0.660635	3.303173
6	0.488385	2.930309	0.533757	3.202540	0.621486	3.728917
7	0.438733	3.071133	0.472328	3.306297	0.543974	3.807820
8	0.395296	3.162369	0.421271	3.370165	0.487759	3.902072
9	0.355389	3.198497	0.388533	3.496801	0.444391	3.999521
10	0.320749	3.207489	0.372467	3.724666	0.412608	4.126081

Table 2: Comparison of OMP Performance Metrics for Different K Values

Comparison of Hybrid Performance Metrics for Different K Values						
Number of Processes	**K=3**		**K=6**		**K=9**	
	Efficiency	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**
2	0.413280	1.653122	0.472820	1.891281	0.469430	1.877720
3	0.328071	1.968425	0.407913	2.447479	0.393274	2.359646
4	0.256843	2.054740	0.336392	2.691139	0.346129	2.769033
5	0.222209	2.222093	0.283367	2.833668	0.290411	2.904108
6	0.197058	2.364694	0.244586	2.935037	0.262185	3.146216
7	0.193476	2.708670	0.221002	3.094025	0.239894	3.358517
8	0.181218	2.899495	0.202122	3.233948	0.227926	3.646820
9	0.169539	3.051707	0.192833	3.471000	0.213811	3.848605
10	0.161493	3.229867	0.186070	3.721408	0.213087	4.261750

Table 3: Comparison of Hybrid Performance Metrics for Different K Values

4.7 Dataset

The initial dataset used as the basis for the research is the Iris dataset, one of the best known and most used datasets in the field of machine learning and data mining. The Iris dataset contains measurements on flowers of three iris species: Iris setosa, Iris versicolor and Iris virginica. For each flower, length and width measurements of the sepals and petals are provided.

However, despite the popularity of the Iris dataset, its limited size (130) can limit the effectiveness of analyzes on it, especially in contexts where greater computational complexity is required or one wants to study large-scale data.

To overcome this limitation, a Python script was developed to generate a new dataset containing over a million data points. This approach facilitated the creation of a larger and more diverse dataset, thereby enhancing the ability to evaluate the effectiveness of the approach.

In the first approach, as mentioned before, a dataset of over a million points was generated. However, despite the dataset's size, the timings did not meet the criteria for a proper benchmark. Therefore, to achieve better results, a dataset of 100 million points was generated to demonstrate the improvements resulting from parallelization.

Furthermore, given the importance of the parameter K in the k-means algorithm, The dataset was extended by generating several versions with varying values of K, including values like 3, 6, and 9. This approach allowed exploration of how data dimensionality influences the outcomes of the k-means algorithm and assessed its adaptability to diverse scenarios.

This idea of exploring variations in the behavior of the k-means algorithm by modifying the value of K derives from similar approaches described in other scientific papers, which have highlighted the importance of examining the behavior of the algorithm as the parameters vary to better understand the its performance and limitations.

5 Analysis

As indicated in the previous paragraphs, numerous tests of the algorithm were conducted both to obtain an average value of the timings (occasionally the cluster results were affected by delays or inaccuracies) and to observe how the metrics vary with changes in K (an essential parameter for a k-means algorithm). As can be observed from Figure 1 (the graphs refer to the scenario with K equal to 3), the trend of the curve for both speedup and efficiency reflects the expected behavior resulting from proper parallelization. Indeed, with the increase in the number of processes/threads used for parallelization, the efficiency value decreases while the speedup value increases. This correct behavior is observable in both the MPI approach, the OMP approach, and the hybrid version.

The performance metrics, Speedup and Efficiency, provide insightful comparisons between the three parallelization approaches: MPI, OMP, and the hybrid MPI+OMP.

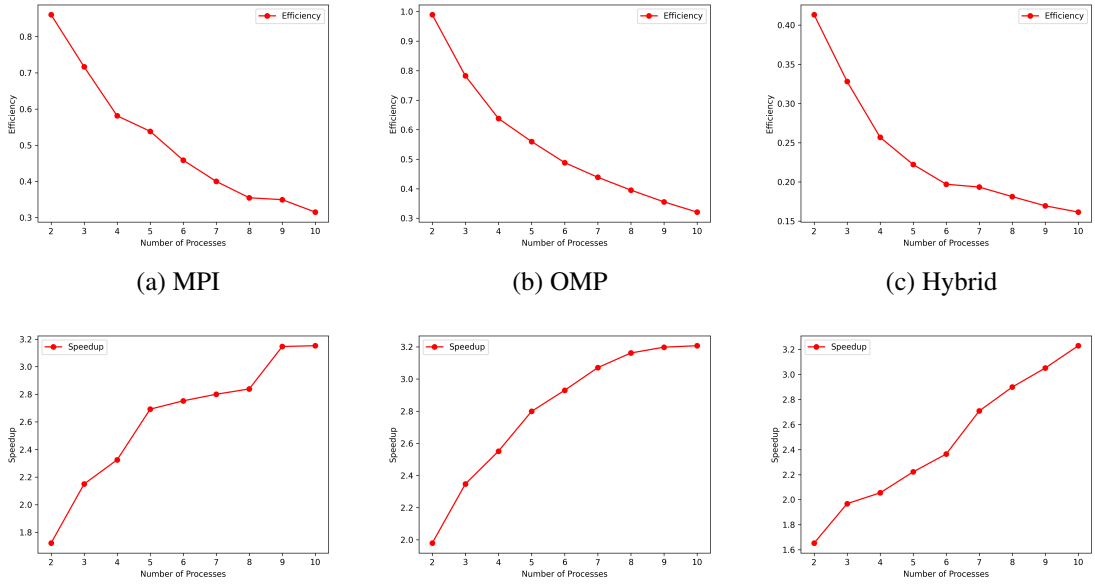
From Tables 4, 5 and 6, it is evident that the Speedup achieved by the OMP approach consistently surpasses that of MPI across all tested values of K (3, 6, and 9). This indicates that OMP is more effective in reducing the execution time of the k-means algorithm compared to MPI. The Speedup values for OMP are higher, demonstrating its superior performance in utilizing multiple cores on a shared-memory system.

However, the hybrid approach, which combines both MPI and OMP, shows a slight improvement in Speedup compared to OMP alone (Table 4 and Table 6). This suggests that leveraging the strengths of both distributed memory (MPI) and shared memory (OMP) parallelization can provide additional performance benefits. The hybrid approach effectively balances the workload across multiple nodes and cores, leading to better overall performance.

Regarding Efficiency, the results show that all three approaches maintain values within the interval [0,1], which is expected. Efficiency generally decreases as the number of processes increases, due to the overhead associated with parallelization. In the case of MPI, the Efficiency values drop more rapidly compared to OMP, reflecting the communication overhead between processes in a distributed memory system. OMP, on the other hand, shows higher Efficiency values, indicating better utilization of computational resources within a shared-memory architecture.

The hybrid approach also maintains Efficiency within the acceptable range but tends to have lower values compared to OMP. This can be attributed to the additional complexity of managing both inter-node communication (MPI) and intra-node threading (OMP). Nevertheless, the hybrid approach still provides a viable option for optimizing performance, especially for large datasets where the combined strengths of MPI and OMP can be fully leveraged.

In summary, while OMP offers better Speedup and Efficiency for smaller numbers of processes due to lower overhead in shared-memory systems, the hybrid approach demonstrates potential for improved performance by combining the advantages of both MPI and OMP. MPI alone is less efficient due to higher communication costs, but it remains essential for scenarios requiring extensive distribution of computational tasks across multiple nodes.



(a) MPI (b) OMP (c) Hybrid

(d) MPI (e) OMP (f) Hybrid

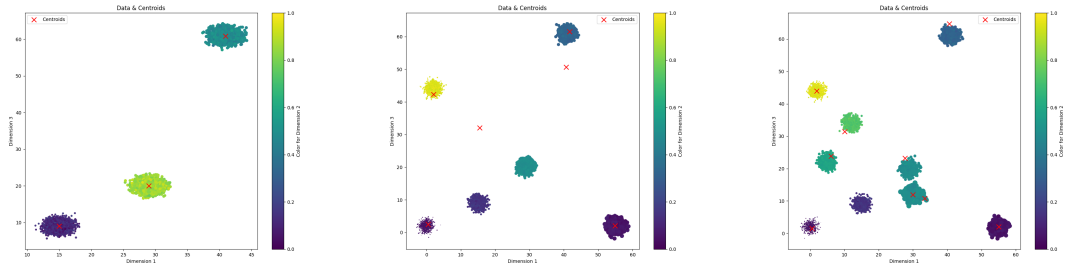
Figure 1: Collection of graphs showing the trends of the metrics, Efficiency (a, b and c) and Speedup (d, e and f), for the dataset with K = 3.

As for the clustering predictions, it has been observed that the resulting centroids remain the same regardless of the number of processes/threads used. Therefore, we can conclude that parallelization is an approach that does not affect the execution of the algorithm itself but only its duration.

In addition to calculating the execution time, we can also evaluate the performance of the algorithm. Although the k-means algorithm does not allow for analytical computation of precision, clustering quality can still be assessed because the number of centroids in the dataset is predetermined. This known information allows us to compare the final centroids produced by the algorithm with the original centroids. Further-

more, Figure 2 provides a visual representation of the clustering by plotting the data points and the resulting centroids, allowing us to visually assess whether the clustering was successful.

The augmented version of the Ibris dataset used contains four features related to the petals, making graphical visualization of centroids more complex in its three-dimensional representation where one feature must be omitted. This omission can lead to potential misinterpretations due to incomplete feature representation. However, considering the omitted feature as a reference point, clusters initially appearing misclassified may actually be correctly classified when all four features are taken into account.



(a) (b) (c)

Figure 2: Collection of images showing the resulting centroids and data in different scenarios: (a) K = 3, (b) K = 6, and (c) K = 9. Furthermore, to clearly visualize the data and reduce the computational load for visualization, reduced datasets were used.

Number of Processes	Efficiency			Speedup		
	MPI	OMP	Hybrid	MPI	OMP	Hybrid
2	0.860483	0.989542	0.413280	1.720966	1.979084	1.653122
3	0.716549	0.782414	0.328071	2.149648	2.347241	1.968425
4	0.581231	0.637823	0.256843	2.324923	2.551294	2.054740
5	0.538460	0.559797	0.222209	2.692300	2.798987	2.222093
6	0.458777	0.488385	0.197058	2.752661	2.930309	2.364694
7	0.400060	0.438733	0.193476	2.800418	3.071133	2.708670
8	0.354858	0.395296	0.181218	2.838863	3.162369	2.899495
9	0.349623	0.355389	0.169539	3.146608	3.198497	3.051707
10	0.315235	0.320749	0.161493	3.152354	3.207489	3.229867

Table 4: Comparison of Efficiency and Speedup for K=3 across MPI, OMP, and Hybrid approaches. The best values are highlighted.

Number of Processes	Efficiency			Speedup		
	MPI	OMP	Hybrid	MPI	OMP	Hybrid
2	0.862711	0.998924	0.472820	1.725423	1.997847	1.891281
3	0.727735	0.825223	0.407913	2.183205	2.475669	2.447479
4	0.623718	0.751991	0.336392	2.494871	3.007964	2.691139
5	0.554441	0.627570	0.283367	2.772206	3.137852	2.833668
6	0.495604	0.533757	0.244586	2.973626	3.202540	2.935037
7	0.452750	0.472328	0.221002	3.169251	3.306297	3.094025
8	0.414930	0.421271	0.202122	3.319439	3.370165	3.233948
9	0.382743	0.388533	0.192833	3.444690	3.496801	3.471000
10	0.350588	0.372467	0.186070	3.505878	3.724666	3.721408

Table 5: Comparison of Efficiency and Speedup for K=6 across MPI, OMP, and Hybrid approaches. The best values are highlighted.

Number of Processes	Efficiency			Speedup		
	MPI	OMP	Hybrid	MPI	OMP	Hybrid
2	0.729479	0.970196	0.469430	1.458958	1.940392	1.877720
3	0.603513	0.815737	0.393274	1.810539	2.447212	2.359646
4	0.552503	0.698491	0.346129	2.210012	2.793966	2.769033
5	0.580493	0.660635	0.290411	2.902466	3.303173	2.904108
6	0.501304	0.621486	0.262185	3.007824	3.728917	3.146216
7	0.459731	0.543974	0.239894	3.218116	3.807820	3.358517
8	0.430395	0.487759	0.227926	3.443158	3.902072	3.646820
9	0.399318	0.444391	0.213811	3.593862	3.999521	3.848605
10	0.373334	0.412608	0.213087	3.733344	4.126081	4.261750

Table 6: Comparison of Efficiency and Speedup for K=9 across MPI, OMP, and Hybrid approaches. The best values are highlighted.

6 Conclusion

To conclude, it appears that the parallelization of the k-means clustering algorithm has been developed correctly. As mentioned earlier, the recorded timings demonstrate a consistent trend with varying numbers of processes/threads used, the calculated metrics are coherent, and the clustering algorithm results are valid. However, despite recording promising results, the algorithm shows some limitations in some aspects: first of all, the datasets were generated in such a way as to have the data well separated and therefore it is not very generalizable. Finally, a limitation and a potential future implementation concern the parallelization of the data reading and writing phase from files. During development, issues were encountered, and ultimately, the decision was made to keep these phases external to the parallelization.

References

[1] Zahid Ansari Ansari Abdullah Quazi Mateenuddin H. “An OpenMP Based Approach for Parallelization and Performance Evaluation of k-Means Algorithm”. In: 2021.

[2] D S. Bhupal Naik, S. Deva Kumar, and S. V. Ramakrishna. “Parallel processing of enhanced K-means using OpenMP”. In: *2013 IEEE International Conference on Computational Intelligence and Computing Research*. 2013, pp. 1–4. DOI: 10.1109/ICCIC.2013.6724291.

[3] Sakshi Patel, Shivani Sihmar, and Aman Jatain. “A study of hierarchical clustering algorithms”. In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, pp. 537–541.

[4] G. Wu et al. “A Parallel K-Means Clustering Algorithm with MPI”. In: *Parallel Architectures, Algorithms and Programming, International Symposium on*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2011, pp. 60–64. DOI: 10.1109/PAAP.2011.17. URL: https://doi.ieeecomputersociety.org/10.1109/PAAP.2011.17.

Algorithm 1 Sequential K-Means Algorithm

```
1:  procedure KMEANS(dataset, K, max_iter)
2:      centroids ← INITIALIZECENTROIDS(dataset, K)
3:      iter ← 0
4:      while iter < max_iter do
5:          for i in 0 to ROWS − 1 do
6:              min_distance ← DBL_MAX
7:              min_index ← 0
8:              for j in 0 to K − 1 do
9:                  distance ← EUCLIDEANDISTANCE(i, j)
10:                 if distance < min_distance then
11:                     min_distance ← distance
12:                     min_index ← j
13:                 end if
14:             end for
15:             if cluster[i] ≠ min_index then
16:                 cluster[i] ← min_index
17:             end if
18:         end for
19:         sum[K][COLS] ← 0
20:         count[K] ← 0
21:         for i in 0 to ROWS − 1 do
22:             for j in 0 to COLS − 1 do
23:                 sum[cluster[i]][j] += dataset[i][j]
24:             end for
25:             count[cluster[i]] += 1
26:         end for
27:         for i in 0 to K − 1 do
28:             for j in 0 to COLS − 1 do
29:                 if count[i] ≠ 0 then
30:                     centroids[i][j] ← sum[i][j]/count[i]
31:                 end if
32:             end for
33:         end for
34:         iter ← iter + 1
35:     end while
36:     WRITECENTROIDSTOFILE(centroids, K)
37: end procedure
```

Algorithm 2 Parallel K-Means Algorithm with MPI

procedure KMEANSMPI($dataset$, K, max_iter)
  $centroids \leftarrow$ INITIALIZECENTROIDS($dataset$, K)
  $t1 \leftarrow$ MPI_WTIME
  $t\_per\_iter \leftarrow 0$
  $iter \leftarrow 0$
  while $iter < max\_iter$ **do**
    for i in 0 to $ROWS - 1$ **do**
      $min\_distance \leftarrow$ DBL_MAX
      $min\_index \leftarrow 0$
      for j in 0 to $K - 1$ **do**
        $distance \leftarrow$ EUCLIDEANDISTANCE(i, j)
        if $distance < min\_distance$ **then**
          $min\_distance \leftarrow distance$
          $min\_index \leftarrow j$
        end if
      end for
      if $cluster[i] \neq min\_index$ **then**
        $cluster[i] \leftarrow min\_index$
      end if
    end for
    $local\_sum[K][COLS] \leftarrow 0$
    $local\_count[K] \leftarrow 0$
    for i in $start$ to $end - 1$ **do**
      for j in 0 to $COLS - 1$ **do**
        $local\_sum[cluster[i]][j] \mathrel{+}= dataset[i][j]$
      end for
      $local\_count[cluster[i]] \mathrel{+}= 1$
    end for
    MPI_BARRIER(MPI_COMM_WORLD)
    MPI_ALLREDUCE($local\_sum$, $global\_sum$, $K * COLS$, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
    MPI_ALLREDUCE($local\_count$, $global\_count$, K, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
    for i in 0 to $K - 1$ **do**
      for j in 0 to $COLS - 1$ **do**
        if $global\_count[i] \neq 0$ **then**
          $centroids[i][j] \leftarrow global\_sum[i][j]/global\_count[i]$
        end if
      end for
    end for
    $iter \leftarrow iter + 1$
  end while
  WRITECENTROIDSTOFILE($centroids$, K)
  $t2 \leftarrow$ MPI_WTIME
  $t\_per\_iter \leftarrow (t2 - t1)/max\_iter$
  MPI_ALLREDUCE(t_per_iter, $sum\_iter\_kmeans\_time$, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
end procedure

Algorithm 3 OpenMP K-Means Algorithm

```
1: procedure KMEANSOMP(dataset, K, max_iter, sum_iter_kmeans_time)
2:     initialize centroids randomly
3:     iter ← 0
4:     time1, time2 ← 0
5:     count[K] ← {0}
6:     sum[K][COLS] ← {{0}}
7:     parallel region
8:     while iter < max_iter do
9:         time1 ← OMP_GET_WTIME
10:        reset count, sum
11:        parallel for
12:        for i in 0 to ROWS − 1 do
13:            min_distance ← DBL_MAX
14:            min_index ← 0
15:            for j in 0 to K − 1 do
16:                distance ← EUCLIDEANDISTANCE(i, j)
17:                if distance < min_distance then
18:                    min_distance ← distance
19:                    min_index ← j
20:                end if
21:            end for
22:            if cluster[i] ≠ min_index then
23:                cluster[i] ← min_index
24:            end if
25:            count[cluster[i]] ← count[cluster[i]] + 1
26:            for j in 0 to COLS − 1 do
27:                sum[cluster[i]][j] ← sum[cluster[i]][j] + dataset[i][j]
28:            end for
29:        end for
30:        combine count and sum across threads
31:        calculate new centroids
32:        single
33:        for i in 0 to K − 1 do
34:            if count[i] ≠ 0 then
35:                for j in 0 to COLS − 1 do
36:                    centroids[i][j] ← sum[i][j]/count[i]
37:                end for
38:            end if
39:        end for
40:        iter ← iter + 1
41:        time2 ← (omp_get_wtime() − time1)/iter
42:        *sum_iter_kmeans_time ← *sum_iter_kmeans_time + time2
43:    end while
44:    write centroids to file
45: end procedure
```